Apache ZooKeeper

深入浅出

作者：Eddie Chueng

Table of Contents

自序	1.1
概述	1.2
安装和运行Zookeeper	1.3
Zookeeper开发实例	1.4
ZooKeeper中的组和成员	1.4.1
创建组	1.4.2
加入组	1.4.3
成员列表	1.4.4
删除分组	1.4.5
Zookeeper 服务	1.5
数据模型 Data Model	1.5.1
操作 Operations	1.5.2
实现 Implementation	1.5.3
数据一致性 Consistency	1.5.4
会话 Sessions	1.5.5
ZooKeeper应用程序 Building Applications with ZooKeeper	1.6
配置服务 Configuration Service	1.6.1
坚韧的ZooKeeper应用 The Resilient ZooKeeper Application	1.6.2
一个稳定的配置服务 A reliable configuration service	1.6.3
生产环境中的ZooKeeper ZooKeeper in Production	1.7
韧性和性能 Resilience and Performance	1.7.1
配置	1.7.2

自序

2016年9月底，于北京

最早接触ZooKeeper是因为工作上使用了Kafka集群，看了一些ZooKeeper的资料。那时对ZooKeeper懵懵懂懂，后来心里有很多疑问。比如，那时根本搞不清ZooKeeper是如何实现集群调度的，client的意义和znode的意义是什么？leader选举是ZooKeeper服务器之间的策略算法，还是Client之间的策略算法，甚至当时我眞的混淆了（书中也提到了这一点）？ZooKeeper到底起了什么作用？而且最开始的一个错误理解，认为每一台Kafka服务器上都需要一个ZooKeeper，然后ZooKeeper来帮助Kafka实现集群内的数据一致性等特性。

当时按照教程搭建了Kafka集群，应用起来也是行云流水，好像跟ZooKeeper没有多大关系了。后来，我买了本《Hadoop: The Definitive Guide 4th Edition》。当通读了ZooKeeper相关章节后，我的心里对我之前ZooKeeper的认识，只剩下两个字了——呵呵-_-|||

一切都好像拨云见日那样清爽了。不仅仅是对ZooKeeper的认识更深了，也让自己对分布式系统的认识上升了一个台阶。

最后，我考虑要整理一下关于ZooKeeper的读书笔记，其实内容多是读书时自己的翻译。那为什么不把《Hadoop: The Definitive Guide 4th Edition》关于ZooKeeper的内容翻译过来呢，这样不是对于我来说更简单一些？于是，我就在我的博客上开始了翻译工作。

经过两个月陆陆续续的翻译，现在终于可以发出来了！本书的内容来自《Hadoop: The Definitive Guide 4th Edition》，在这里向书的作者和贡献者致以崇高的敬意。

本书的内容纯属个人业余翻译，欢迎各位读者批评！这里留下作者的email，欢迎大家吐槽！**holynull@126.com**

概述

Zookeeper是Hadoop分布式调度服务，用来构建分布式应用系统。构建一个分布式应用是一个很复杂的事情，主要的原因是我们需要合理有效的处理分布式集群中的部分失败的问题。例如，集群中的节点在相互通信时，A节点向B节点发送消息。A节点如果想知道消息是否发送成功，只能由B节点告诉A节点。那么如果B节点关机或者由于其他的原因脱离集群网络，问题就出现了。A节点不断的向B发送消息，并且无法获得B的响应。B也没有办法通知A节点已经离线或者关机。集群中其他的节点完全不知道B发生了什么情况，还在不断的向B发送消息。这时，你的整个集群就发生了部分失败的故障。

Zookeeper不能让部分失败的问题彻底消失，但是它提供了一些工具能够让你的分布式应用安全合理的处理部分失败的问题。

安装和运行Zookeeper

我们采用standalone模式，安装运行一个单独的zookeeper服务。安装前请确认您已经安装了Java运行环境。

我们去Apache ZooKeeper releases page下载zookeeper安装包，并解压到本地：

```
% tar xzf zookeeper-x.y.z.tar.gz
```

ZooKeeper提供了一些可执行程序的工具，为了方便起见，我们将这些工具的路径加入到PATH环境变量中：

```
% export ZOOKEEPER_HOME=~/sw/zookeeper-x.y.z
% export PATH=$PATH:$ZOOKEEPER_HOME/bin
```

运行ZooKeeper之前我们需要编写配置文件。配置文件一般在安装目录下的`conf/zoo.cfg`。我们可以把这个文件放在`/etc/zookeeper`下，或者放到其他目录下，并在环境变量设置`ZOOCFGDIR`指向这个个目录。下面是配置文件的内容：

```
tickTime=2000
dataDir=/Users/tom/zookeeper
clientPort=2181
```

tickTime是zookeeper中的基本时间单元，单位是毫秒。datadir是zookeeper持久化数据存放的目录。clientPort是zookeeper监听客户端连接的端口，默认是2181.

启动命令：

```
% zkServer.sh start
```

我们通过`nc`或者`telnet`命令访问`2181`端口，通过执行ruok（Are you OK?）命令来检查zookeeper是否启动成功：

```
% echo ruok | nc localhost 2181
imok
```

那么我看见zookeeper回答我们“I’m OK”。下表中是所有的zookeeper的命名，都是由4个字符组成。

Category	Command	Description
Server status	ruok	Prints imok if the server is running and not in an error state.
	conf	Prints the server configuration (from zoo.cfg).
	envi	Prints the server environment, including ZooKeeper version, Java version, and other system properties.
	srvr	Prints server statistics, including latency statistics, the number of znodes, and the server mode (standalone, leader, or follower).
	stat	Prints server statistics and connected clients.
	srst	Resets server statistics.
	isro	Shows whether the server is in read-only (ro) mode (due to a network partition) or read/write mode (rw).
Client connections	dump	Lists all the sessions and ephemeral znodes for the ensemble. You must connect to the leader (see srvr) for this command.
	cons	Lists connection statistics for all the server's clients.
	crst	Resets connection statistics.
Watches	wchs	Lists summary information for the server's watches.
	wchc	Lists all the server's watches by connection. Caution: may impact server performance for a large number of watches.
	wchp	Lists all the server's watches by znode path. Caution: may impact server performance for a large number of watches.
Monitoring	mntr	Lists server statistics in Java properties format, suitable as a source for monitoring systems such as Ganglia and Nagios.

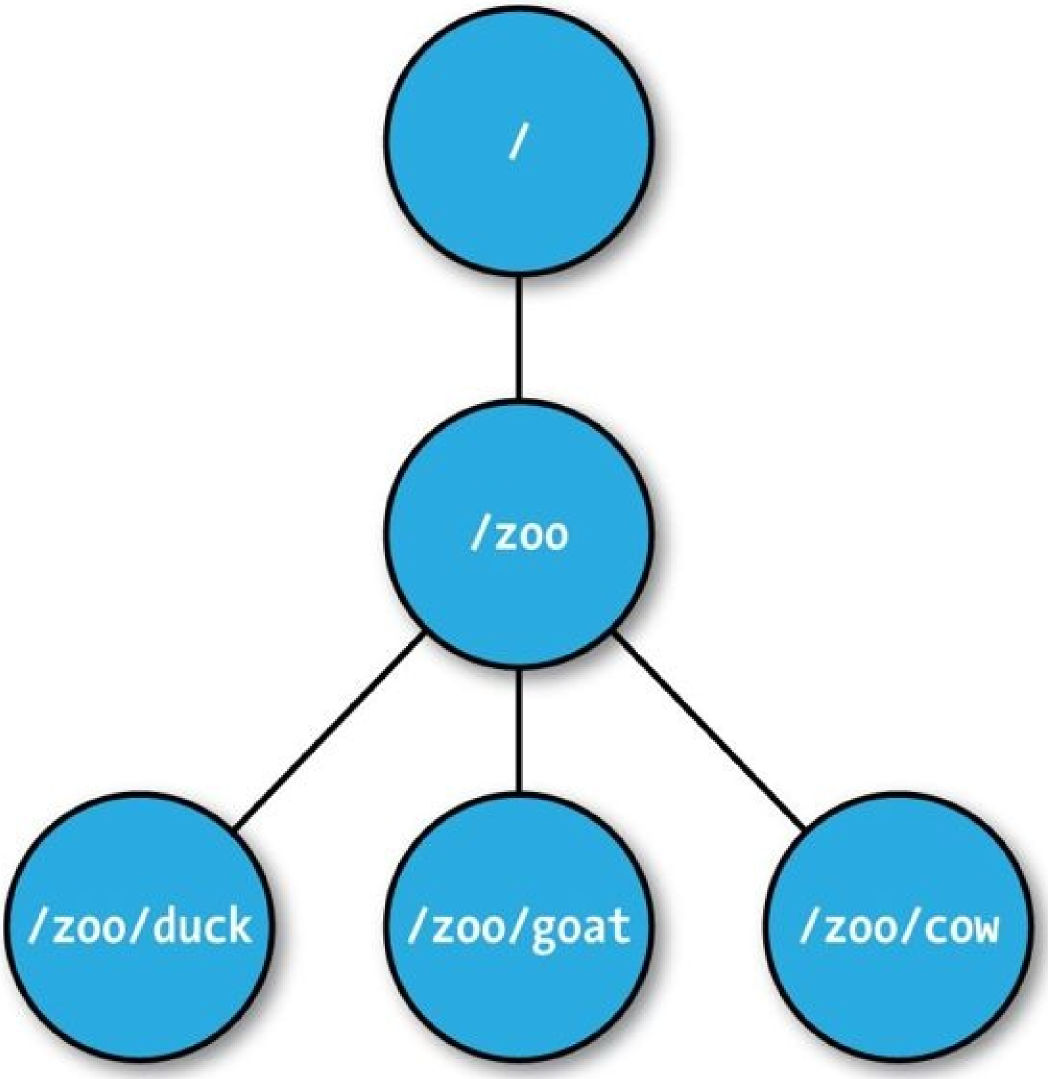
3.5.0以上的版本会有一个内嵌的web服务，通过访问`http://localhost:8080/commands`来访问以上的命令列表。

Zookeeper开发实例

这一节我们将讲解如何编写Zookeeper客户端的程序，来控制zookeeper上的数据，以达到管理客户端所在集群的成员关系。

ZooKeeper中的组和成员

我们可以把Zookeeper理解为一个高可用的文件系统。但是它没有文件和文件夹的概念，只有一个叫做znode的节点概念。那么znode即是数据的容器，也是其他节点的容器。（其实znode就可以理解为文件或者是文件夹）我们用父节点和子节点的关系来表示组和成员的关系。那么一个节点代表一个组，组节点下的子节点代表组内的成员。如下图所示：



创建组

我们使用zookeeper的Java API来创建一个 `/zoo` 的组节点：

```
public class CreateGroup implements Watcher {
 private static final int SESSION_TIMEOUT = 5000;
 private ZooKeeper zk;
 private CountDownLatch connectedSignal = new CountDownLatch(1);
 public void connect(String hosts) throws IOException, InterruptedException {
     zk = new ZooKeeper(hosts, SESSION_TIMEOUT, this);
     connectedSignal.await();
 }

 @Override
 public void process(WatchedEvent event) { // Watcher interface
     if (event.getState() == KeeperState.SyncConnected) {
     connectedSignal.countDown();
     }
 }

 public void create(String groupName) throws KeeperException,
 InterruptedException {
     String path = "/" + groupName;
     String createdPath = zk.create(path, null/*data*/, Ids.OPEN_ACL_UNSAFE,
     CreateMode.PERSISTENT);
     System.out.println("Created " + createdPath);
 }

 public void close() throws InterruptedException {
     zk.close();
 }

 public static void main(String[] args) throws Exception {
     CreateGroup createGroup = new CreateGroup();
     createGroup.connect(args[0]);
     createGroup.create(args[1]);
     createGroup.close();
     }
 }
```

当 `main()` 执行时，首先创建了一个 `CreateGroup` 的对象，然后调用 `connect()` 方法，通过zookeeper的API与zookeeper服务器连接。创建连接我们需要3个参数：一是服务器端主机名称以及端口号，二是客户端连接服务器session的超时时间，三是Watcher接口的一个实例。Watcher实例负责接收Zookeeper数据变化时产生的事件回调。

在连接函数中创建了zookeeper的实例，然后建立与服务器的连接。建立连接函数会立即返回，所以我们需要等待连接建立成功后再进行其他的操作。我们使用CountDownLatch来阻塞当前线程，直到zookeeper准备就绪。这时，我们就看到Watcher的作用了。我们实现了Watcher接口的一个方法：

```
public void process(WatchedEvent event);
```

当客户端连接上了zookeeper服务器，Watcher将由 `process()` 函数接收一个连接成功的事件。我们接下来调用CountDownLatch，释放之前的阻塞。

连接成功后，我们调用 `create()` 方法。我们在这个方法中调用zookeeper实例的 `create()` 方法来创建一个znode。参数包括：一是znode的path；二是znode的内容（一个二进制数组），三是一个access control list(ACL，访问控制列表，这里使用完全开放模式)，最后是znode的性质 。

znode的性质分为ephemeral和persistent两种。ephemeral性质的znode在创建他的客户端的会话结束，或者客户端以其他原因断开与服务器的连接时，会被自动删除。而persistent性质的znode就不会被自动删除，除非客户端主动删除，而且不一定是创建它的客户端可以删除它，其他客户端也可以删除它。这里我们创建一个persistent的znode。

`create()` 将返回znode的path。我们讲新建znode的path打印出来。

我们执行如上程序：

```
% export CLASSPATH=ch21-zk/target/classes/:$ZOOKEEPER_HOME/*:\
$ZOOKEEPER_HOME/lib/*:$ZOOKEEPER_HOME/conf
% java CreateGroup localhost zoo
Created /zoo
```

加入组

接下来我们实现如何在一个组中注册成员。我们将使用ephemeral znode来创建这些成员节点。那么当客户端程序退出时，这些成员将被删除。

我们创建一个ConnetionWatcher类，然后继承实现一个JoinGroup类：

```
public class ConnectionWatcher implements Watcher {

 private static final int SESSION_TIMEOUT = 5000;

 protected ZooKeeper zk;

 private CountDownLatch connectedSignal = new CountDownLatch(1);

 public void connect(String hosts) throws IOException, InterruptedException {
     zk = new ZooKeeper(hosts, SESSION_TIMEOUT, this);
     connectedSignal.await();
 }

 @Override
 public void process(WatchedEvent event) {
     if (event.getState() == KeeperState.SyncConnected) {
         connectedSignal.countDown();
     }
 }

 public void close() throws InterruptedException {
     zk.close();
 }
}
```

```
public class JoinGroup extends ConnectionWatcher {

 public void join(String groupName, String memberName) throws KeeperException,
 InterruptedException {
     String path = "/" + groupName + "/" + memberName;
     String createdPath = zk.create(path, null/*data*/, Ids.OPEN_ACL_UNSAFE,
     CreateMode.EPHEMERAL);
     System.out.println("Created " + createdPath);
 }

 public static void main(String[] args) throws Exception {
     JoinGroup joinGroup = new JoinGroup();
     joinGroup.connect(args[0]);
     joinGroup.join(args[1], args[2]);
     // stay alive until process is killed or thread is interrupted
     Thread.sleep(Long.MAX_VALUE);
 }
}
```

加入组与创建组非常相似。我们加入了一个ephemeral znode后，让线程阻塞住。然后我们可以使用命令行查看zookeeper中我们创建的znode。当我们将阻塞的程序强行关闭后，我们会发现我们创建的znode会自动消失。

成员列表

下面我们实现一个程序来列出一个组中的所有成员。

```
public class ListGroup extends ConnectionWatcher {

  public void list(String groupName) throws KeeperException,
  InterruptedException {
      String path = "/" + groupName;
      try {
          List<String> children = zk.getChildren(path, false);
          if (children.isEmpty()) {
              System.out.printf("No members in group %s\n", groupName);
              System.exit(1);
          }
          for (String child : children) {
              System.out.println(child);
          }
      } catch (KeeperException.NoNodeException e) {
          System.out.printf("Group %s does not exist\n", groupName);
          System.exit(1);
      }
  }

  public static void main(String[] args) throws Exception {
      ListGroup listGroup = new ListGroup();
      listGroup.connect(args[0]);
      listGroup.list(args[1]);
      listGroup.close();
  }
}
```

我们在 `list()` 方法中通过调用 `getChildren()` 方法来获得某一个path下的子节点，然后打印出来。我们这里会试着捕获KeeperException.NoNodeException，当znode不存在时会抛出这个异常。我们运行程序，会看见如下结果，说明我们还没在zoo组中添加任何成员几点：

```
% java ListGroup localhost zoo
No members in group zoo
```

我们可以运行之前的 `JoinGroup` 来添加成员。在后台运行一些JoinGroup程序，这些程序添加节点后都处于sleep状态：

```
% java JoinGroup localhost zoo duck &
% java JoinGroup localhost zoo cow &
% java JoinGroup localhost zoo goat &
% goat_pid=$!
```

最后一行命令的作用是将最后一个启动的java程序的pid记录下来，我们好在列出zoo下面的成员后，将该进程kill掉。

下面我们将zoo下的成员打印出来：

```
% java ListGroup localhost zoo
goat
duck
cow
```

然后我们将kill掉最后启动的JoinGroup客户端：

```
% kill $goat_pid
```

过几秒后，我们发现goat节点不见了。因为之前我们创建的goat节点是一个ephemeral节点，而创建这个节点的客户端在ZooKeeper上的会话已经被终结了，因为这个回话在5秒后失效了（我们设置了会话的超时时间为5秒）：

```
% java ListGroup localhost zoo
duck
cow
```

让我们回过头来看看，我们到底都做了一些什么？我们首先创建了一个节点组，这些节点的创建者都在同一个分布式系统中。这些节点的创建者之间互相都不知情。一个创建者想使用这些节点数据进行一些工作，例如通过znode节点是否存在来判断节点的创建者是否存在。

最后一点，我们不能只依靠组成员关系来完全解决在与节点通信时的网络错误。当与一个集群组成员节点进行通信时，发生了通信失败，我们需要使用重试或者试验与组中其他的节点通信，来解决这次通信失败。

Zookeeper的命令行工具

Zookeeper有一套命令行工具。我们可以像如下使用，来查找zoo下的成员节点：

```
% zkCli.sh -server localhost ls /zoo
[cow, duck]
```

你可以不加参数运行这个工具，来获得帮助。

删除分组

下面让我们来看一下如何删除一个分组？

ZooKeeper的API提供一个 `delete()` 方法来删除一个znode。我们通过输入znode的path和版本号（version number）来删除想要删除的znode。我们除了使用path来定位我们要删除的znode，还需要一个参数是版本号。只有当我们指定要删除的本版号，与znode当前的版本号一致时，ZooKeeper才允许我们将znode删除掉。这是一种optimistic locking机制，用来处理znode的读写冲突。我们也可以忽略版本号一致检查，做法就是版本号赋值为-1。

删除一个znode之前，我们需要先删除它的子节点，就下如下代码中实现的那样：

```
public class DeleteGroup extends ConnectionWatcher {

 public void delete(String groupName) throws KeeperException,
 InterruptedException {
     String path = "/" + groupName;

     try {
         List<String> children = zk.getChildren(path, false);
         for (String child : children) {
             zk.delete(path + "/" + child, -1);
         }
         zk.delete(path, -1);
     } catch (KeeperException.NoNodeException e) {
         System.out.printf("Group %s does not exist\n", groupName);
         System.exit(1);
     }
 }

 public static void main(String[] args) throws Exception {
     DeleteGroup deleteGroup = new DeleteGroup();
     deleteGroup.connect(args[0]);
     deleteGroup.delete(args[1]);
     deleteGroup.close();
 }
}
```

最后我们执行如下操作来删除zoo group：

```
% java DeleteGroup localhost zoo
% java ListGroup localhost zoo
Group zoo does not exist
```

Zookeeper 服务

ZooKeeper 是一个高可用的高性能调度服务。这一节我们将讲述他的模型、操作和接口。

数据模型 Data Model

ZooKeeper包含一个树形的数据模型，我们叫做znode。一个znode中包含了存储的数据和ACL（Access Control List）。ZooKeeper的设计适合存储少量的数据，并不适合存储大量数据，所以znode的存储限制最大不超过1M。

数据的访问被定义成原子性的。什么是原子性呢？一个客户端访问一个znode时，不会只得到一部分数据；客户端访问数据要么获得全部数据，要么读取失败，什么也得不到。相似的，写操作时，要么写入全部数据，要么写入失败，什么也写不进去。ZooKeeper能够保证写操作只有两个结果，成功和失败。绝对不会出现只写入了一部分数据的情况。与HDFS不同，ZooKeeper不支持字符的append（连接）操作。原因是HDFS是被设计成支持数据流访问（streaming data access）的大数据存储，而ZooKeeper则不是。

我们可以通过path来定位znode，就像Unix系统定位文件一样，使用斜杠来表示路径。但是，znode的路径只能使用绝对路径，而不能想Unix系统一样使用相对路径，即Zookeeper不能识别 `../` 和 `./` 这样的路径。

节点的名称是由Unicode字符组成的，除了 `zookeeper` 这个字符串，我们可以任意命名节点。为什么不能使用 `zookeeper` 命名节点呢？因为ZooKeeper已经默认使用 `zookeeper` 来命名了一个根节点，用来存储一些管理数据。

请注意，这里的path并不是URIs，在Java API中是一个String类型的变量。

Ephemeral znodes

我们已经知道，znode有两种类型：ephemeral和persistent。在创建znode时，我们指定znode的类型，并且在之后不会再被修改。当创建znode的客户端的session结束后，ephemeral类型的znode将被删除。persistent类型的znode在创建以后，就与客户端没什么联系了，除非主动去删除它，否则他会一直存在。Ephemeral znode没有任何子节点。

虽然Ephemeral znode绑定了客户端session，但是对任何其他客户端都是可见的，当然是在他们的ACL策略下允许访问的情况下。

当我们在创建分布式系统时，需要知道分布式资源是否可用。Ephemeral znode就是为这种场景应运而生的。正如我们之前讲述的例子中，使用Ephemeral znode来实现一个成员关系管理，任何一个客户端进程任何时候都可以知道其他成员是否可用。

Znode的序号

如果在创建znode时，我们使用排序标志的话，ZooKeeper会在我们指定的znode名字后面增加一个数字。我们继续加入相同名字的znode时，这个数字会不断增加。这个序号的计数器是由这些排序znode的父节点来维护的。

如果我们请求创建一个znode，指定命名为 `/a/b-` ，那么ZooKeeper会为我们创建一个名字为 `/a/b-3` 的znode。我们再请求创建一个名字为 `/a/b-` 的znode，ZooKeeper会为我们创建一个名字 `/a/b-5` 的znode。ZooKeeper给我们指定的序号是不断增长的。Java API中的 `create()` 的返回结果就是znode的实际名字。

那么序号用来干什么呢？当然是用来排序用的！后面《A Lock Service》中我们将讲述如何使用znode的序号来构建一个share lock。

观察模式 Watches

观察模式可以使客户端在某一个znode发生变化时得到通知。观察模式有ZooKeeper服务的某些操作启动，并由其他的一些操作来触发。例如，一个客户端对一个znode进行了 `exists` 操作，来判断目标znode是否存在，同时在znode上开启了观察模式。如果znode不存在，这 `exists` 将返回 `false` 。如果稍后，另外一个客户端创建了这个znode，观察模式将被触发，将znode的创建事件通知之前开启观察模式的客户端。我们将在以后详细介绍其他的操作和触发。

观察模式只能被触发一次。如果要一直获得znode的创建和删除的通知，那么就需要不断的在znode上开启观察模式。在上面的例子中，如果客户端还继续需要获得znode被删除的通知，那么在获得创建通知后，客户端还需要继续对这个znode进行 `exists` 操作，再开启一次观察模式。

在《A Configuration Service》中，有一个例子将讲述如何使用观察模式在集群中更新配置。

操作 Operations

下面的表格中列出了9种ZooKeeper的操作。

操作	说明
create	Creates a znode (the parent znode must already exist)
delete	Deletes a znode (the znode must not have any children)
exists	Tests whether a znode exists and retrieves its metadata
getACL, setACL	Gets/sets the ACL for a znode
getChildren	Gets a list of the children of a znode
getData, setData	Gets/sets the data associated with a znode
sync	Synchronizes a client’s view of a znode with ZooKeeper

调用 `delete` 和 `setData` 操作时，我们必须指定一个znode版本号（version number），即我们必须指定我们要删除或者更新znode数据的哪个版本。如果版本号不匹配，操作将会失败。失败的原因可能是在我们提交之前，该znode已经被修改过了，版本号发生了增量变化。那么我们该怎么办呢？我可以考虑重试，或者调用其他的操作。例如，我们提交更新失败后，可以重新获取znode当前的数据，看看当前的版本号是什么，再做更新操作。

ZooKeeper虽然可以被看作是一个文件系统，但是由于ZooKeeper文件很小，所以没有提供像一般文件系统所提供的 `open` 、 `close` 或者 `seek` 操作。

注意
这里的 `sync` 操作与POSIX文件系统的 `fsync()` 操作是不同的。就像我们早前讲过的，ZooKeeper的写操作是原子性的，一个成功的写操作只保证数据被持久化到大多数ZooKeeper的服务器存储上。所以读操作可能会读取不到最新状态的数据， `sync` 操作用来让client强制所访问的ZooKeeper服务器上的数据状态更新到最新状态。我们会在《一致性 Consistentcy》一节中详细介绍。

批量更新 Multiupdate

ZooKeeper支持将一些原始的操作组合成一个操作单元，然后执行这些操作。那么这种批量操作也是具有原子性的，只可能有两种执行结果，成功和失败。批量操作单元中的操作，不会出现一些操作执行成功，一些操作执行失败的情况，即要么都成功，要么都失败。

Multiupdate对于绑定一些结构化的全局变量很有用处。例如绑定一个无向图（undirected graph）。无向图的顶点（vertex）由znode来表示。添加和删除边（edge）的操作，由修改边的两个关联znode来实现。如果我们使用ZooKeeper的原始的操作来实现对边（edge）的操

作，那么就有可能产生两个znode修改不一致的情况（一个修改成功，一个修改失败）。那么我们将修改两个znode的操作放入到一个Multi修改单元中，就能够保证两个znode，要么都修改成功，要么都修改失败。这样就能够避免修改无向图的边时产生修改不一致的现象。

APIs

ZooKeeper客户端使用的核心编程语言有JAVA和C；同时也支持Perl、Python和REST。执行操作的方式呢，分为同步执行和异步执行。我们之前已经见识过了同步的Java API中的`exists`。

```
public Stat exists(String path, Watcher watcher) throws KeeperException,
 InterruptedException
```

下面代码则是异步方式的`exists`:

```
public void exists(String path, Watcher watcher, StatCallback cb, Object ctx)
```

Java API中，异步的方法的返回类型都是`void`，而操作的返回的结果将传递到回调对象的回调函数中。回调对象将实现`StatCallback`接口中的一个回调函数，来接收操作返回的结果。函数接口如下：

```
public void processResult(int rc, String path, Object ctx, Stat stat);
```

参数`rc`表示返回码，请参考`KeeperException`中的定义。在`stat`参数为null的情况下，非0的值表示一种异常。参数`path`和`ctx`与客户端调用的`exists`方法中的参数相等，这两个参数通常用来确定回调中获得的响应是来至于哪个请求的。参数`ctx`可以是任意对象，只有当`path`参数不能消灭请求的歧义时才会用到。如果不需要参数`ctx`，可以设置为null。

应该使用同步**API**还是异步**API**呢**?**
两种API提供了相同的功能，需要使用哪种API取决于你程序的模式。例如，你设计的程序模式是一个事件驱动模式的程序，那么你最好使用异步API。异步API也可以被用在追求一个比较好的数据吞吐量的场景。想象一下，如果你需要得去大量的znode数据，并且依靠独立的进程来处理他们。如果使用同步API,每次读取操作都会被阻塞住，直到返回结果。不如使用异步API，读取操作可以不必等待返回结果，继续执行。而使用另外的线程来处理返回结果。

观察模式触发器 Watch triggers

读操作，例如：`exists`、`getChildren`、`getData`会在znode上开启观察模式，并且写操作会触发观察模式事件，例如：`create`、`delete`和`setData`。ACL(Access Control List)操作不会启动观察模式。观察模式被触发时，会生成一个事件，这个事件的类型取决于触发他的

操作：

- `exists`启动的观察模式，由创建znode，删除znode和更新znode操作来触发。
- `getData`启动的观察模式，由删除znode和更新znode操作触发。创建znode不会触发，是因为`getData`操作成功的前提是znode必须已经存在。
- `getChildren`启动的观察模式，由子节点创建和删除，或者本节点被删除时才会被触发。我们可以通过事件的类型来判断是本节点被删除还是子节点被删除：`NodeChildrenChanged`表示子节点被删除，而`NodeDeleted`表示本节点删除。

---	Watch trigger			
Watch creation	create znode	create child	delete znode	delete child
exists	NodeCreated	-	NodeDeleted	-
getData	-	-	NodeDeleted	-
getChildren	-	getChildren	NodeDeleted	NodeChildrenChanged

事件包含了触发事件的znode的path，所以我们通过`NodeCreated`和`NodeDeleted`事件就可以知道哪个znode被创建了或者删除了。如果我们需要在`NodeChildrenChanged`事件发生后知道哪个子节点被改变了，我们就需要再调用一次`getChildren`来获得一个新的子节点列表。与之类似，在`NodeDataChanged`事件发生后，我们需要调用`getData`来获得新的数据。我们在编写程序时，会在接收到事件通知后改变znode的状态，所以我们一定要清楚的记住znode的状态变化。

ACLs 访问控制操作

znode的创建时，我们会给他一个ACL（Access Control List），来决定谁可以对znode做哪些操作。

ZooKeeper通过鉴权来获得客户端的身份，然后通过ACL来控制客户端的访问。鉴权方式有如下几种：

- digest

  使用用户名和密码方式

- sasl

  使用Kerberos鉴权

- ip

  使用客户端的IP来鉴权

客户端可以在与ZooKeeper建立会话连接后，自己给自己授权。授权是并不是必须的，虽然znode的ACL要求客户端必须是身份合法的，在这种情况下，客户端可以自己授权来访问znode。下面的例子，客户端使用用户名和密码为自己授权：

```
zk.addAuthInfo("digest", "tom:secret".getBytes());
```

ACL是由鉴权方式、鉴权方式的ID和一个许可（permession）的集合组成。例如，我们想通过一个ip地址为10.0.0.1的客户端访问一个znode。那么，我们需要为znode设置一个ACL，鉴权方式使用IP鉴权方式，鉴权方式的ID为10.0.0.1，只允许读权限。使用JAVA我们将像如下方式创建一个ACL对象：

```
new ACL(Perms.READ,new Id("ip", "10.0.0.1"));
```

所有的许可权限将在下表中列出。请注意，`exists`操作不受ACL的控制，所以任何一个客户端都可以通过`exists`操作来获得任何znode的状态，从而得知znode是否真的存在。

ACL permission	Permitted operations
CREATE	create (a child znode)
READ	getChildren，getData
WRITE	setData
DELETE	delete (a child znode)
ADMIN	setACL

在`ZooDefs.Ids`类中，有一些ACL的预定义变量，包括`OPEN_ACL_UNSAFE`，这个设置表示将赋予所有的许可给客户端（除了ADMIN的许可）。

另外，我们可以使用ZooKeeper鉴权的插件机制，来整合第三方的鉴权系统。

实现 Implementation

ZooKeeper服务可以在两种模式下运行。在standalone模式下，我们可以运行一个单独的ZooKeeper服务器，我们可以在这种模式下进行基本功能的简单测试，但是这种模式没有办法体现ZooKeeper的高可用特性和快速恢复特性。在生产环境中，我们一般采用replicated（复制）模式安装在多台服务器上，组建一个叫做ensemble的集群。ZooKeeper在他的副本之间实现高可用性，并且只要ensemble集群中能够推举出主服务器，ZooKeeper的服务就可以一直不终断。例如，在一个5个节点的ensemble中，容忍有2个节点脱离集群，服务还是可用的。因为剩下的3个节点投票，可以产生超过集群半数的投票，来推选一台主服务器。而6个节点的ensemble中，也只能容忍2个节点的服务器死机。因为如果3个节点脱离集群，那么剩下的3个节点无论如何不能产生超过集群半数的投票来推选一个主服务器。所以，一般情况下ensemble中的服务器数量都是奇数。

从概念上来看，ZooKeeper其实是很简单的。他所做的一切就是保证每一次对znode树的修改，都能够复制到ensemble的大多数服务器上。如果非主服务器脱离集群，那么至少有一台服务器上的副本保存了最新状态。剩下的其他的服务器上的副本，会很快更新这个最新的状态。

为了实现这个简单而不平凡的设计思路，ZooKeeper使用了一个叫做Zab的协议。这个协议分为两阶段，并且不断的运行在ZooKeeper上：

- 阶段 1：领导选举（Leader election）

Ensemble中的成员通过一个程序来选举出一个首领成员，我们叫做leader。其他的成员就叫做follower。在大多数（quorum）follower完成与leader状态同步时，这个阶段才结束。

- 阶段 2： 原子广播（Atomic broadcast）

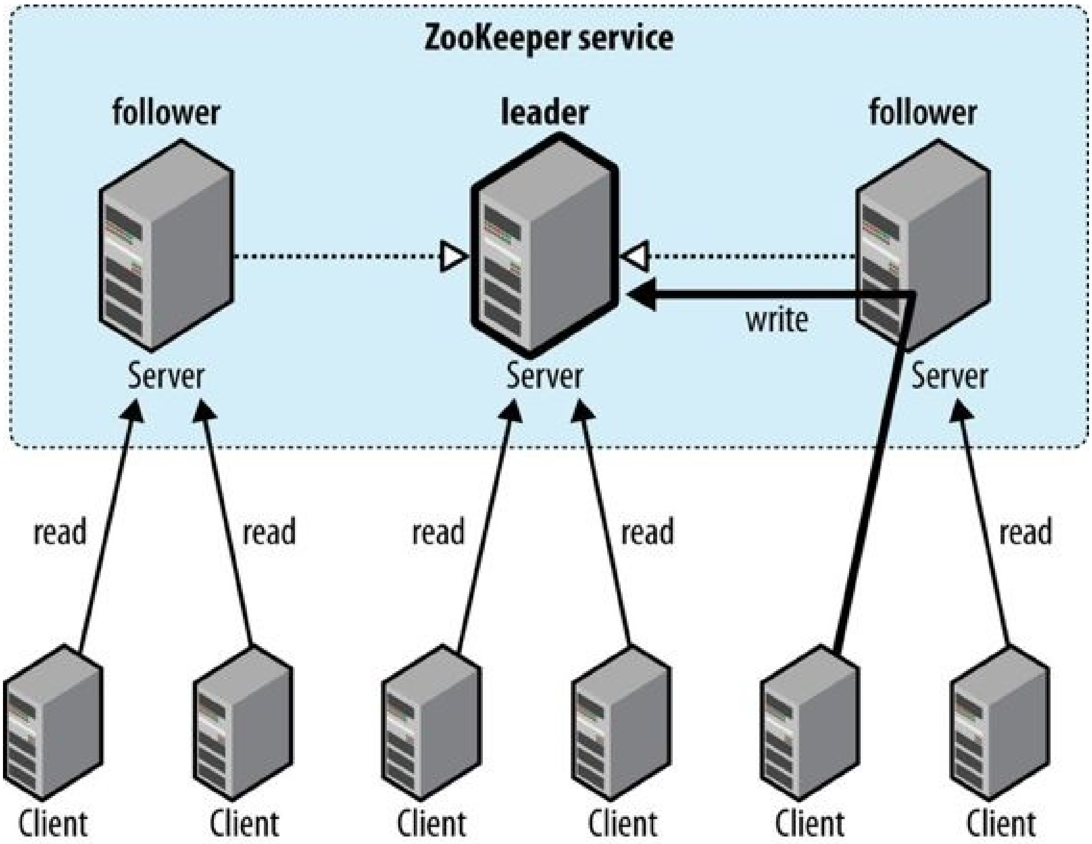
所有的写入请求都会发送给leader，leader在广播给follower。当大多数的follower已经完成了数据改变，leader才会将更新提交，客户端就会随之得到leader更新成功的消息。协议中的设计也是具有原子性的，所以写入操作只有成功和失败两个结果。

如果leader脱离了集群，剩下的节点将选举一个新的leader。如果之前的leader回到了集群中，那么将被视作一个follower。leader的选举很快，大概200ms就能够产生结果，所以不会影响执行效率。

Ensemble中的所有节点都会在更新内存中的znode树的副本之前，先将更新数据写入到硬盘上。读操作可以请求任何一台ZooKeeper服务器，而且读取速度很快，因为读取是内存中的数据副本。

数据一致性 Consistency

理解了ZooKeeper的实现原理，有助于理解ZooKeeper如何保证数据的一致性。就像字面上理解的“leader”和“follower”的意思一样，在ensemble中follower的update操作会滞后于leader的update完成。事实的结果使我们在提交更新数据之前，不必在每一台ZooKeeper服务器上执行持久化变更数据，而是仅需在主服务器上执行持久化变更数据。ZooKeeper客户端的最佳实践是全部链接到follower上。然而客户端是有可能连接到leader上的，并且客户端控制不了这个选择，甚至客户端并不知道连接到了follower还是leader。下图所示，读操作向follower请求即可，而写操作由leader来提交。



每一个对znode树的更新操作，都会被赋予一个全局唯一的ID，我们称之为zxid（ZooKeeper Transaction ID）。更新操作的ID按照发生的时间顺序升序排序。例如，z_1小于z_2，那么z_1的操作就早于z_2操作。

ZooKeeper在数据一致性上实现了如下几个方面：

- 顺序一致性

  从客户端提交的更新操作是按照先后循序排序的。例如，如果一个客户端将一个znode z赋值为a，然后又将z的值改变成b，那么在这个过程中不会有客户端在z的值变为b后，取到的值是a。

- 原子性

  更新操作的结果不是失败就是成功。即，如果更新操作失败，其他的客户端是不会知道的。

- 系统视图唯一性

  无论客户端连接到哪个服务器，都将看见唯一的系统视图。如果客户端在同一个会话中去连接一个新的服务器，那么他所看见的视图的状态不会比之前服务器上看见的更旧。当ensemble中的一个服务器宕机，客户端去尝试连接另外一台服务器时，如果这台服务器的状态旧于之前宕机的服务器，那么服务器将不会接受客户端的连接请求，直到服务器的状态赶上之前宕机的服务器为止。

- 持久性

  一旦更新操作成功，数据将被持久化到服务器上，并且不能撤销。所以服务器宕机重启，也不会影响数据。

- 时效性

  系统视图的状态更新的延迟时间是有一个上限的，最多不过几十秒。如果服务器的状态落后于其他服务器太多，ZooKeeper会宁可关闭这个服务器上的服务，强制客户端去连接一个状态更新的服务器。

从执行效率上考虑，读操作的目标是内存中的缓存数据，并且读操作不会参与到写操作的全局排序中。这就会引起客户端在读取ZooKeeper的状态时产生不一致。例如，A客户端将znode z的值由a改变成a'，然后通知客户端B去读取z的值，但是B读取到的值是a，而不是修改后的a'。为了阻止这种情况出现，B在读取z的值之前，需要调用 `sync` 方法。 `sync` 方法会强制B连接的服务器状态与leader的状态同步，这样B在读取z的值就是A重新更改过的值了。

注意
`sync` 操作只在异步调用时才可用，原因是你不需要等待操作结束再去执行其他的操作。因此，ZooKeeper保证所有的子操作都会在 `sync` 结束后再执行，甚至在 `sync` 操作之前发出的操作请求也不例外。

会话 Sessions

ZooKeeper的客户端中，配置了一个ensemble服务器列表。当启动时，首先去尝试连接其中一个服务器。如果尝试连接失败，那么会继续尝试连接下一个服务器，直到连接成功或者全部尝试连接失败。

一旦连接成功，服务器就会为客户端创建一个会话（session）。session的过期时间由创建会话的客户端应用来设定，如果在这个时间期间，服务器没有收到客户端的任何请求，那么session将被视为过期，并且这个session不能被重新创建，而创建的ephemeral znode将随着session过期被删除掉。在会话长期存在的情况下，session的过期事件是比较少见的，但是应用程序如何处理好这个事件是很重要的。（我们将在《The Resilient ZooKeeper Application》中详细介绍）

在长时间的空闲情况下，客户端会不断的发送ping请求来保持session。（ZooKeeper的客户端开发工具的liberay实现了自动发送ping请求，所以我们不必去考虑如何维持session）ping请求的间隔被设置成足够短，以便能够及时发现服务器失败（由读操作的超时时长来设置），并且能够及时的在session过期前连接到其他服务器上。

容错连接到其他服务器上，是由ZooKeeper客户端自动完成的。重要的是在连接到其他服务器上后，之前的session以及epemeral节点还保持可用状态。

在容错的过程中，应用将收到与服务断开连接和连接的通知。Watch模式的通知在断开链接时，是不会发送断开连接事件给客户端的，断开连接事件是在重新连接成功后发送给客户端的。如果在重新连接到其他节点时，应用尝试一个操作，这个操作是一定会失败的。对于这一点的处理，是一个ZooKeeper应用的重点。（我们将在《The Resilient ZooKeeper Application》中讲述）

时间 Time

在ZooKeeper中有一些时间的参数。`tick`是ZooKeeper的基础时间单位，用来定义ensemble中服务器上运行的程序的时间表。其他时间相关的配置都是以`tick`为单位的，或者以`tick`的值为最大值或者最小值。例如，session的过期时间在2 ticks到20 ticks之间，那么你再设置时选择的session过期时间必须在2和20之间的一个数。

通常情况1 tick等于2秒。那么就是说session的过期时间的设置范围在4秒到40秒之间。在session过期时间的设置上有一些考虑。过期时间太短会造成加快物理失败的监测频率。在组成员关系的例子中，session的过期时间与从组中移除失败的成员花费的时间相等。如果设置过低的session过期时间，那么网络延迟就有可能造成非预期的session过期。这种情况下，就会出现在短时间内一台机器不断的离开组，然后又从新加入组中。

如果应用需要创建比较复杂的临时状态，那么就需要较长的session过期时间，因为重构花费的时间比较长。有一些情况下，需要在session的生命周期内重启，而且要保证重启完后session不过期（例如，应用维护和升级的情况）。服务器会给每一个session一个ID和密码，如果在连接创建时，ZooKeeper验证通过，那么session将被恢复使用（只要session没过期就行）。所以应用程序可以实现一个优雅的关机动作，在重启之前，将session的ID和密码存储在一个稳定的地方。重启之后，通过ID和密码恢复session。

这仅仅是在一些特殊的情况下，我们需要使用这个特性来使用比较长的session过期时间。大多数情况下，我们还是要考虑当出现非预期的异常失败时，如何处理session过期，或者仅需要优雅的关闭应用，在session过期前不用重启应用。

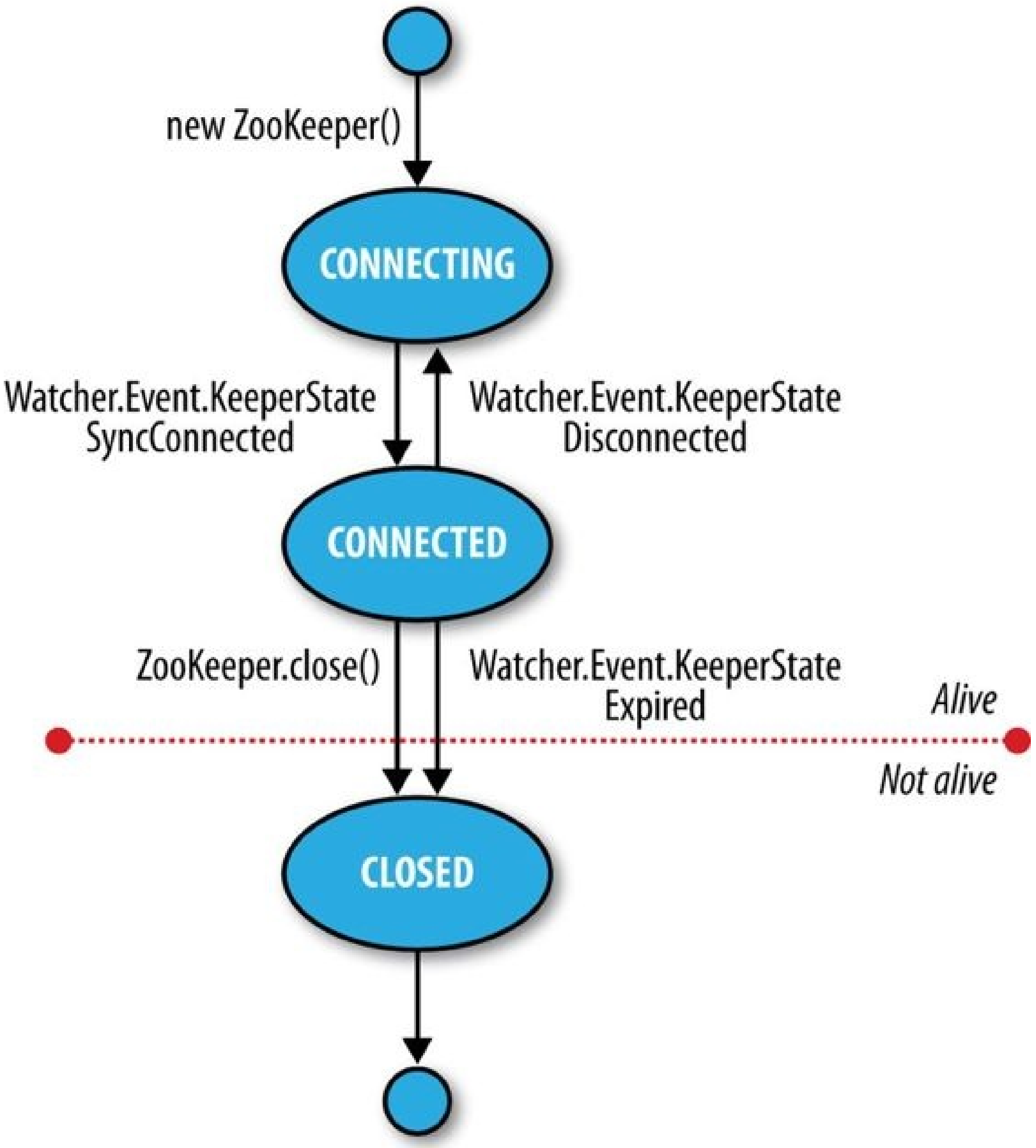
通常情况也越大规模的ensemble，就需要越长的session过期时间。Connetction Timeout、Read Timeout和Ping Periods都由一个以服务器数量为参数的函数计算得到，当ensemble的规模扩大，这些值需要逐渐减小。如果为了解决经常失去连接而需要增加timeout的时长，建议你先监控一下ZooKeeper的metrics，再去调整。

状态 States

ZooKeeper对象在他的生命周期内会有不同的状态，我们通过 `getState()` 来获得当前的状态。

```
public States getState()
```

状态是一个枚举类型的数据。新构建的ZooKeeper对象在尝试连接ZooKeeper服务时的状态是 `CONNECTING` ，一旦与服务建立了连接那么状态就变成了 `CONNECTED` 。



客户端可以通过注册一个观察者对象来接收ZooKeeper对象状态的迁移。当通过 `CONNECTED` 状态后，观察者将接收到一个WatchedEvent事件，他的属性KeeperState的值是 `SyncConnected` 。

注意
观察者有两个职能：一是接收ZooKeeper的状态改变通知；二是接收znode的改变通知。ZooKeeper对象构造时传递进去的watcher对象，默认是用来接收状态改变通知的，但是znode的改变通知也可能会共享使用默认的watcher对象，或者使用一个专用的watcher。我们可以通过一个Boolean变量来指定是否使用共享默认watcher。

ZooKeeper实例会与服务连接断开或者重新连接，状态会在 `CONNECTING` 和 `CONNECTED` 之间转换。如果连接断开，watcher会收到一个断开连接事件。请注意，这两个状态都是ZooKeeper实例自己初始化的，并且在断开连接后会自动进行重连接。

如果调用了`close()`或者session过期，ZooKeeper实例会转换为第三个状态`CLOSED`，此时在接受事件的KeeperState属性值为`Expired`。一旦ZooKeeper的状态变为`CLOSED`，说明实例已经不可用（可以通过`isAlive()`来判断），并且不能再被使用。如果要重新建立连接，就需要重新构建一个ZooKeeper实例。

ZooKeeper应用程序 Building Applications with ZooKeeper

在对ZooKeeper有了一个深入的了解以后，我们来看一下用ZooKeeper可以实现哪些应用。

配置服务 Configuration Service

一个基本的ZooKeeper实现的服务就是“配置服务”，集群中的服务器可以通过ZooKeeper共享一个通用的配置数据。从表面上，ZooKeeper可以理解为一个配置数据的高可用存储服务，为应用提供检索和更新配置数据服务。我们可以使用ZooKeeper的观察模式实现一个活动的配置服务，当配置数据发生变化时，可以通知与配置相关客户端。

接下来，我们来实现一个这样的活动配置服务。首先，我们设计用znode来存储key-value对，我们在znode中存储一个String类型的数据作为value，用znode的path来表示key。然后，我们实现一个client，这个client可以在任何时候对数据进行跟新操作。那么这个设计的ZooKeeper数据模型应该是：master来更新数据，其他的worker也随之将数据更新，就像HDFS的namenode那样。

我们在一个叫做ActiveKeyValueStore的类中编写代码如下：

```
public class ActiveKeyValueStore extends ConnectionWatcher {

  private static final Charset CHARSET = Charset.forName("UTF-8");

  public void write(String path, String value) throws InterruptedException,
      KeeperException {
    Stat stat = zk.exists(path, false);
    if (stat == null) {
      zk.create(path, value.getBytes(CHARSET), Ids.OPEN_ACL_UNSAFE,
          CreateMode.PERSISTENT);
    } else {
      zk.setData(path, value.getBytes(CHARSET), -1);
    }
  }
}
```

`write()` 方法主要实现将给定的key-value对写入到ZooKeeper中。这其中隐含了创建一个新的znode和更新一个已存在的znode的实现方法的不同。那么操作之前，我们需要根据 `exists()` 来判断znode是否存在，然后再根据情况进行相关的操作。其他值得一提的就是String类型的数据在转换成 `byte[]` 时，使用的字符集是UTF-8。

我们为了说明 `ActiveKeyValueStore` 怎么使用，我们考虑实现一个 `ConfigUpdater` 类来实现更新配置。下面代码实现了一个在一些随机时刻更新配置数据的应用。

```
public class ConfigUpdater {

  public static final String PATH = "/config";

  private ActiveKeyValueStore store;
  private Random random = new Random();

  public ConfigUpdater(String hosts) throws IOException, InterruptedException {
    store = new ActiveKeyValueStore();
    store.connect(hosts);
  }

  public void run() throws InterruptedException, KeeperException {
    while (true) {
      String value = random.nextInt(100) + "";
      store.write(PATH, value);
      System.out.printf("Set %s to %s\n", PATH, value);
      TimeUnit.SECONDS.sleep(random.nextInt(10));
    }
  }

  public static void main(String[] args) throws Exception {
    ConfigUpdater configUpdater = new ConfigUpdater(args[0]);
    configUpdater.run();
  }
}
```

上面的代码很简单。在 `ConfigUpdater` 的构造函数中， `ActiveKeyValueStore` 对象连接到ZooKeeper服务。然后 `run()` 不断的循环运行，使用一个随机数不断的随机更新 `/config` znode上的值。

下面我们来看一下，如何读取 `/config` 上的值。首先，我们在 `ActiveKeyValueStore` 中实现一个读方法。

```
public String read(String path, Watcher watcher) throws InterruptedException,
      KeeperException {
    byte[] data = zk.getData(path, watcher, null/*stat*/);
    return new String(data, CHARSET);
  }
```

ZooKeeper的 `getData()` 方法的参数包含：path，一个Watcher对象和一个Stat对象。Stat对象中含有从 `getData()` 返回的值，并且负责接收回调信息。这种方式下，调用者不仅可以获得数据，还能够获得znode的metadata。

做为服务的consumer， `ConfigWatcher` 以观察者身份，创建一个 `ActiveKeyValueStore` 对象，并且在启动以后调用 `read()` 函数（在 `dispalayConfig()` 函数中）获得相关数据。

下面的代码实现了一个以观察模式获得ZooKeeper中的数据更新的应用，并将值到后台中。

```
public class ConfigWatcher implements Watcher {

  private ActiveKeyValueStore store;

  public ConfigWatcher(String hosts) throws IOException, InterruptedException {
    store = new ActiveKeyValueStore();
    store.connect(hosts);
  }

  public void displayConfig() throws InterruptedException, KeeperException {
    String value = store.read(ConfigUpdater.PATH, this);
    System.out.printf("Read %s as %s\n", ConfigUpdater.PATH, value);
  }

  @Override
  public void process(WatchedEvent event) {
    if (event.getType() == EventType.NodeDataChanged) {
      try {
        displayConfig();
      } catch (InterruptedException e) {
        System.err.println("Interrupted. Exiting.");
        Thread.currentThread().interrupt();
      } catch (KeeperException e) {
        System.err.printf("KeeperException: %s. Exiting.\n", e);
      }
    }
  }

  public static void main(String[] args) throws Exception {
    ConfigWatcher configWatcher = new ConfigWatcher(args[0]);
    configWatcher.displayConfig();

    // stay alive until process is killed or thread is interrupted
    Thread.sleep(Long.MAX_VALUE);
  }
}
```

当 `ConfigUpadater` 更新znode时，ZooKeeper将触发一个 `EventType.NodeDataChanged` 的事件给观察者。 `ConfigWatcher` 将在他的 `process()` 函数中获得这个时间，并将显示读取到的最新的版本的配置数据。

由于观察模式的触发是一次性的，所以每次都要调用 `ActiveKeyValueStore` 的 `read()` 方法，这样才能获得未来的更新数据。我们不能确保一定能够接受到更新通知事件，因为在接受观察事件和下一次读取之间的窗口期内，znode可能被改变了（有可能很多次），但是client可能没有注册观察模式，所以client不会接到znode改变的通知。在配置服务中这不是一个什么问题，因为client只关心配置数据的最新版本。然而，建议读者关注一下这个潜在的问题。

让我们来看一下控制台打印的 `ConfigUpdater` 运行结果：

```
% java ConfigUpdater localhost
Set /config to 79
Set /config to 14
Set /config to 78
```

然后立即在另外的控制台终端窗口中运行 `ConfigWatcher` ：

```
% java ConfigWatcher localhost
Read /config as 79
Read /config as 14
Read /config as 78
```

坚韧的ZooKeeper应用 The Resilient ZooKeeper Application

分布式计算设计的第一谬误就是认为“网络是稳定的”。我们所实现的程序目前都是假设网络稳定的情况下实现的，所以当我们在一个眞实的网络环境下，会有很多原因可以使程序执行失败。下面我们将阐述一些可能造成失败的场景，并且讲述如何正确的处理这些失败，让我们的程序在面对这些异常时更具韧性。

在ZooKeeper的API中，每一个ZooKeeper的操作都会声明抛出两个异常：InterruptedException和KeeperException。

InterrupedException

当一个操作被中断时，会抛出一个InterruptedException。在JAVA中有一个标准的阻塞机制用来取消程序的执行，就是在需要阻塞的的地方调用 `interrupt()` 。如果取消执行成功，会以抛出一个InterruptedException作为结果。ZooKeeper坚持了这个标准，所以我们可以用这种方式来取消client的对ZooKeeper的操作。用到ZooKeeper的类和库需要向上抛出InterruptedException，才能使我们的client实现取消操作。

InterruptedException并不意味着程序执行失败，可能是人为设计中断的，所以在上面配置应用的例子中，当向上抛出InterruptedException时，会引起应用终止。

KeeperException

当ZooKeeper服务器出现错误信号，或者出现了通信方面的问题，就会抛出一个KeeperException。由于错误的不同原因，所以KeeperException有很多子类。例如， `KeeperException.NoNodeException` 当操作一个znode时，而这个znode并不存在，就会抛出这个异常。

每一个之类都有一个异常码作为异常的类型。例如， `KeeperException.NoNodeException` 的异常码就是 `KeeperException.Code.NONODE` (一个枚举值)。

有两种方法来处理KeeperException。一种是直接捕获KeeperException，然后根据异常码进行不同类型异常处理。另一种是捕获具体的子类，然后根据不同类型的异常进行处理。

KeeperException包含了3大类异常。

状态异常 State Exception

当无法操作znode树造成操作失败时，会产生状态异常。通常引起状态异常的原因是有另外的程序在同时改变znode。例如，一个 `setData()` 操作时，会抛出 `KeeperException.BadVersionException` 。因为另外的一个程序已经在 `setData()` 操作之前修改了znode，造成 `setData()` 操作时版本号不匹配了。程序员必须了解，这种情况是很有可能发生的，我们必须靠编写处理这种异常的代码来解决他。

有的一些异常是编写代码时的疏忽造成的，例如 `KeeperException.NoChildrenForEphemeralsException` 。这个异常是当我们给一个enphemeral类型的znode添加子节点时抛出的。

重新获取异常 Recoverable Exception

重新获取异常来至于那些能够获得同一个ZooKeeper session的应用。伴随的表现是抛出 `KeeperException.ConnectionLossException` ，表示与ZooKeeper的连接丢失。ZooKeeper将会尝试重新连接，大多数情况下重新连接都会成功并且能够保证session的完整性。

然而，ZooKeeper无法通知客户端操作由于 `KeeperException.ConnectionLossException` 而失败。这就是一个部分失败的例子。只能依靠程序员编写代码来处理这个不确定性。

在这点上，幂等操作和非幂等操作的差别就会变得非常有用了。一个幂等操作是指无论运行一次还是多次结果都是一样的，例如一个读请求，或者一个不设置任何值得setData操作。这些操作可以不断的重试。

一个非幂等操作不能被不分青红皂白的不停尝试执行，就像一些操作执行一次的效率和执行多次的效率是不同。我们将在之后会讨论如何利用非幂等操作来处理Recovreable Exception。

不能重新获取异常 Unrecoverable exceptions

在一些情况下，ZooKeeper的session可能会变成不可用的——比如session过期，或者因为某些原因session被close掉（都会抛出KeeperException.SessionExpiredException），或者鉴权失败（KeeperException.AuthFailedException）。无论何种情况，ephemeral类型的znode上关联的session都会丢失，所以应用在重新连接到ZooKeeper之前都需要重新构建他的状态。

一个稳定的配置服务 A reliable configuration service

回过头来看一下`ActiveKeyValueStore`中的`write()`方法，其中调用了`exists()`方法来判断znode是否存在，然后决定是创建一个znode还是调用setData来更新数据。

```
public void write(String path, String value) throws InterruptedException,
      KeeperException {
    Stat stat = zk.exists(path, false);
    if (stat == null) {
      zk.create(path, value.getBytes(CHARSET), Ids.OPEN_ACL_UNSAFE,
          CreateMode.PERSISTENT);
    } else {
      zk.setData(path, value.getBytes(CHARSET), -1);
    }
  }
```

从整体上来看，`write()`方法是一个幂等方法，所以我们可以不断的尝试执行它。我们来修改一个新版本的`write()`方法，实现在循环中不断的尝试write操作。我们为尝试操作设置了一个最大尝试次数参数（`MAX_RETRIES`）和每次尝试间隔的休眠(`RETRY_PERIOD_SECONDS`)时长：

```
public void write(String path, String value) throws InterruptedException,
      KeeperException {
    int retries = 0;
    while (true) {
      try {
        Stat stat = zk.exists(path, false);
        if (stat == null) {
          zk.create(path, value.getBytes(CHARSET), Ids.OPEN_ACL_UNSAFE,
              CreateMode.PERSISTENT);
        } else {
          zk.setData(path, value.getBytes(CHARSET), stat.getVersion());
        }
        return;
      } catch (KeeperException.SessionExpiredException e) {
        throw e;
      } catch (KeeperException e) {
        if (retries++ == MAX_RETRIES) {
          throw e;
        }
        // sleep then retry
        TimeUnit.SECONDS.sleep(RETRY_PERIOD_SECONDS);
      }
    }
  }
```

细心的读者可能会发现我们并没有在捕获 `KeeperException.SessionExpiredException` 时继续重新尝试操作，这是因为当session过期后，ZooKeeper会变为 `CLOSED` 状态，就不能再重新连接了。我们只是简单的抛出一个异常，通知调用者去创建一个新的ZooKeeper实例，所以 `write()` 方法可以不断的尝试执行。一个简单的方式来创建一个ZooKeeper实例就是重新new一个 `ConfigUpdater` 实例。

```
public static void main(String[] args) throws Exception {
    while (true) {
      try {
        ResilientConfigUpdater configUpdater =
          new ResilientConfigUpdater(args[0]);
        configUpdater.run();
      } catch (KeeperException.SessionExpiredException e) {
        // start a new session
      } catch (KeeperException e) {
        // already retried, so exit
        e.printStackTrace();
        break;
      }
    }
  }
```

另一个可以替代处理session过期的方法就是使用watcher来监控`Expired`的`KeeperState`，然后重新建立一个连接。这种方法下，我们只需要不断的尝试执行`write()`，如果我们得到了KeeperException.SessionExpiredException`异常，连接最终也会被重新建立起来。那么我们抛开如何从一个过期的session中恢复问题，我们的重点是连接丢失的问题也可以这样解决，只是处理方法不同而已。

注意
我们这里忽略了另外一种情况，在zookeeper实例不断的尝试连接了ensemble中的所有节点后发现都无法连接成功，就会抛出一个IOException，说明所有的集群节点都不可用。而有一些应用被设计为不断的尝试连接，直到ZooKeeper服务恢复可用为止。

这只是一个重复尝试的策略。还有很多的策略，比如指数补偿策略，每次尝试之间的间隔时间会被乘以一个常数，间隔时间会逐渐变长，直到与集群建立连接为止间隔时间才会恢复到一个正常值，来预备一下次连接异常使用。

> 译者：为什么要使用指数补偿策略呢？这是为了避免反复的尝试连接而消耗资源。在一次较短的时间后第二次尝试连接不成功后，延长第三次尝试的等待时间，这期间服务恢复的几率可能会更大。第四次尝试的机会就变小了，从而达到减少尝试的次数。

锁服务 A Lock Service

分布式锁用来为一组程序提供互斥机制。任意一个时刻仅有一个进程能够获得锁。分布式锁可以用来实现大型分布式系统的leader选举算法，即leader就是获取到锁的那个进程。

注意
不要把ZooKeeper的原生leader选举算法和我们这里所说的通用leader选举服务搞混淆了。ZooKeeper的原生leader选举算法并不是公开的算法，并不能向我们这里所说的通用leader选举服务那样，为一个分布式系统提供主进程选举服务。

为了使用ZooKeeper实现分布式锁，我们使用可排序的znode来实现进程对锁的竞争。思路其实很简单：首先，我们需要一个表示锁的znode，获得锁的进程就表示被这把锁给锁定了（命名为，/leader）。然后，client为了获得锁，就需要在锁的znode下创建ephemeral类型的子znode。在任何时间点上，只有排序序号最小的znode的client获得锁，即被锁定。例如，如果两个client同时创建znode `/leader/lock-1` 和 `/leader/lock-2`，所以创建`/leader/lock-1`的client获得锁，因为他的排序序号最小。ZooKeeper服务被看作是排序的权威管理者，因为是由他来安排排序的序号的。

锁可能因为删除了`/leader/lock-1`znode而被简单的释放。另外，如果相应的客户端死掉，使用ephemeral znode的价值就在这里，znode可以被自动删除掉。创建`/leader/lock-2`的client就获得了锁，因为他的序号现在最小。当然客户端需要启动观察模式，在znode被删除时才能获得通知：此时他已经获得了锁。

获得锁的伪代码如下：

1. 在lock的znode下创建名字为 `lock-` 的ephemeral类型znode，并记录下创建的znode的path（会在创建函数中返回）。
2. 获取lock znode的子节点列表，并开启对lock的子节点的watch模式。
3. 如果创建的子节点的序号最小，则再执行一次第2步，那么就表示已经获得锁了。退出。
4. 等待第2步的观察模式的通知，如果获得通知，则再执行第2步。

羊群效应

虽然这个算法是正确的，但是还是有一些问题。第一个问题是羊群效应。试想一下，当有成千成百的client正在试图获得锁。每一个client都对lock节点开启了观察模式，等待lock的子节点的变化通知。每次锁的释放和获取，观察模式将被触发，每个client都会得到消息。那么羊群效应就是指像这样，大量的client都会获得相同的事件通知，而只有很小的一部分client会对事件通知有响应。我们这里，只有一个client将获得锁，但是所有的client都得到了通知。那么这就像在网络公路上撒了把钉子，增加了ZooKeeper服务器的压力。

为了避免羊群效应，通知的范围需要更精准。我们通过观察发现，只有当序号排在当前znode之前一个znode离开时，才有必要通知创建当前znode的client，而不必在任意一个znode删除或者创建时都通知client。在我们的例子中，如果client1、client2和client3创建了znode `/leader/lock-1` 、 `/leader/lock-2` 和 `leader/lock-3` ，client3仅在 `/leader/lock-2` 消失时，才获得通知。而不需要在 `/leader/lock-1` 消失时，或者新建 `/leader/lock-4` 时，获得通知。

重新获取异常 Recoverable Exception

这个锁算法的另一个问题是没有处理当连接中断造成的创建失败。在这种情况下，我们根本就不知道之前的创建是否成功了。创建一个可排序的znode是一个非等幂操作，所以我们不能简单重试，因为如果第一次我们创建成功了，那么第一次创建的znode就成了一个孤立的znode了，将永远不会被删除直到会话结束。

那么问题的关键在于，在重新连接以后，client不能确定是否之前创建过lock节点的子节点。我们在znode的名字中间嵌入一个client的ID，那么在重新连接后，就可以通过检查lock znode的子节点znode中是否有名字包含client ID的节点。如果有这样的节点，说明之前创建节点操作成功了，就不需要再创建了。如果没有这样的节点，那就重新创建一个。

Client的会话ID是一个长整型数据，并且在ZooKeeper中是唯一的。我们可以使用会话的ID在处理连接丢失事件过程中作为client的id。在ZooKeeper的JAVA API中，我们可以调用 `getSessionId()` 方法来获得会话的ID。

那么Ephemeral类型的可排序znode不要命名为 `lock-<sessionId>-` ，所以当加上序号后就变成了 `lock-<sessionId>-<sequenceNumber>` 。那么序号虽然针对上一级名字是唯一的，但是上一级名字本身就是唯一的，所以这个方法既可以标记znode的创建者，也可以实现创建的顺序排

序。

不能恢复异常 Unrecoverable Exception

如果client的会话过期，那么他创建的ephemeral znode将被删除，client将立即失去锁（或者至少放弃获得锁的机会）。应用需要意识到他不再拥有锁，然后清理一切状态，重新创建一个锁对象，并尝试再次获得锁。注意，应用必须在得到通知的第一时间进行处理，因为应用不知道如何在znode被删除事后判断是否需要清理他的状态。

实现 Implementation

考虑到所有的失败模式的处理的繁琐，所以实现一个正确的分布式锁是需要做很多细微的设计工作。好在ZooKeeper为我们提供了一个 产品级质量保证的锁的实现，我们叫做**WriteLock**。我们可以轻松的在client中应用。

更多的分布式数据结构和协议 More Distribute Data Structures and Protocols

我们可以用ZooKeeper来构建很多分布式数据结构和协议，例如，barriers，queues和two-phase commit。有趣的是我们注意到这些都是同步协议，而我们却使用ZooKeeper的原生异步特征（比如通知机制）来构建他们。

在ZooKeeper官网上提供了一些数据结构和协议的伪代码。并且提供了实现这些的数据结构和协议的标准教程（包括locks、leader选举和队列）；你可以在**recipes**目录中找到。

Apache Curator project也提供了一些简单客户端的教程。

生产环境中的ZooKeeper ZooKeeper in Production

在生产环境中，你需要在replicated模式下运行ZooKeeper。这里我们将介绍一些运行ZooKeeper服务器集群的注意事项。这里我们只做一个简单介绍，如果你需要了解更详细的内容，请参考《ZooKeeper Administrator’s Guide》。

韧性和性能 Resilience and Performance

ZooKeeper应用应该被定位用于减少机器和网络对系统的影响。在实践中这意味着我将隔离机架、电源供应和路由，使得我们不会因为他们的故障而导致失去我们的大多数服务器。

低延迟服务应用的重点是要求所有的服务器都在一个数据中心里。然而一些不要求低延迟应答的场景，为了获得额外的韧性，将服务器部署在不同的数据中心（至少每两台在一个数据中心）。本节中的例子是一个leader选举算法和一个分布式锁算法，两者都不具有频繁的状态改变的特征，几十毫秒的开销对于系统并不会造成重要的影响。

注意
ZooKeeper的概念中有一类不参加leader选举投票的follower。由于在众多的读请求过程中，这种观察者节点并不参加投票，所以可以提高ZooKeeper集群的读取性能，而不去伤害到写入性能。观察者节点可以部署在跨数据中心的环境下，而不会像参加投票的follower那样在跨数据中心的环境中会对集群产生潜在的影响。那么我们可以将参加投票的follower部署在同一个数据中心，而将不参加投票的follower部署在另外一个数据中心。

ZooKeeper是一个高可用的系统，他的重点是能够及时运行它的功能。因此，建议ZooKeeper服务器最好专注于运行ZooKeeper。如果运行了其他的应用程序，可能会降低ZooKeeper的性能。

配置保证ZooKeeper的事务日志在与他的快照不同的硬盘上。默认情况下，都在 `dataDir` 指定的目录下，我们可以通过额外设置 `dataLogDir` 来指定日志的目录。日志被指定写到专门的硬盘设备，ZooKeeper就可以对大化写日志的速率。

我们在配置文件夹下的 `java.env` 中可以配置JVM参数。

配置

集群中的ZooKeeper服务器都有一个数值ID，范围在1~255之间。这个ID存在 `dataDir` 目录下的 `myid` 文件中。

每一个server必须知道其他的ZooKeeper server在网络中的位置，所以我们需要将所有的server都配置在文件中：

```
server.n=hostname:port:port
```

下面是一个配置例子：

```
tickTime=2000
dataDir=/disk1/zookeeper
dataLogDir=/disk2/zookeeper
clientPort=2181
initLimit=5
syncLimit=2
server.1=zookeeper1:2888:3888
server.2=zookeeper2:2888:3888
server.3=zookeeper3:2888:3888
```

replicated模式下有两个额外参数：

initLimit：follower连接和同步leader的时长。如果大多数follower这个时长内同步失败，将重新选举一个leader代替之前的leader。如果经常发生这种情况，说明这个值设置的太低。

syncLimit：folloer同步leader的时长。如果follower在这个时长内同步失败，follower将自动重启。连接他的client将连接到其他的follower上。